Karlsruhe Reports in Informatics 2014,6
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Detecting Data-Flow Errors in BPMN 2.0

Silvia von Stackelberg, Susanne Putze, Jutta Mülle, and Klemens Böhm

2014

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Detecting Data-Flow Errors in BPMN 2.0

Silvia von Stackelberg, Susanne Putze, Jutta Mülle, and Klemens Böhm

Karlsruhe Institute of Technology, KIT, 76131 Karlsruhe, Germany
{silvia.stackelberg|susanne.putze|jutta.muelle|klemens.boehm}@kit.edu

Abstract. Data-flow errors in BPMN 2.0 process models, such as missing or unused data, lead to undesired process executions. The BPMN 2.0 execution semantics and certain features of the data make it difficult to detect data-flow errors. In particular, BPMN 2.0 allows to specify alternatives for data as well as optional data. In this paper, we propose an approach for detecting data-flow errors in BPMN 2.0 process models. We formalize BPMN process models by mapping them to Petri Nets and unfolding the execution semantics regarding data. We define a set of anti-patterns representing data-flow errors of BPMN 2.0 process models. By employing the anti-patterns, our tool performs model checking for the unfolded Petri Nets. The evaluation shows that it detects all data-flow errors identified by hand, and so improves process quality.

1 Introduction

The language BPMN (Business Process Model and Notation), now an ISO standard, is widely accepted in research and practice. One important trait of BPMN 2.0 is the possibility to specify executable processes. Further, the data aspect of processes gets more attention with BPMN 2.0 [13]. Process designers can now model, for example, the data needs and data results of a task. The data needs describe the data elements a task requires for its execution, data results the ones available afterwards. This does not only hold for tasks, but - with restrictions - also for other flow elements, such as events or conditional sequence flows. Finally, one can specify alternatives for data as well as optional data.

Process designers model the *data flow*, i.e., how data traverses a process, by specifying the data needs and data results for individual flow elements. Typically, a designer starts to define data associations in the graphical process diagram and then refines the model with non graphical specifications for execution, adding information such as alternatives for data or the optional use of data. Ideally, a BPMN 2.0 modelling tool supports these steps.

The problem studied here is to decide whether the data-flow specifications in executable BPMN process models are correct. In line with other research [10,16,20], a data flow is correct if there are no anomalies regarding processed data. Such anomalies occur if data needed by flow elements is not available, if flow elements do not use data produced before, or if they use data inconsistently. Data-flow correctness is crucial. To illustrate, a *missing data* element (e.g., a non-initialized element) may lead to blocking due to starvation, or to incorrect

gateway decisions [3,16]. In executable BPMN models, specifications for optional data and alternatives for data can contain errors as well.

Example 1. Think of a process withdrawing money from an ATM with two alternative authentication methods. In this process, a task *authentication* needs for its execution either the data elements cash card and PIN, or the elements cash card and fingerprint. The error *missing data* occurs if one of the alternative data sets cannot be available at task execution time.

In consequence, an approach for detecting data-flow errors in BPMN 2.0 process models has to take alternatives for data and optional data into account. It is advantageous to check the correctness of a data flow already at design time (*correctness at design*). Existing approaches, e.g., Meda et al. [10], Sadiq et al. [16], Trcka et al. [20], do not take the specifics of BPMN 2.0 into account, namely the execution semantics when using mandatory and optional data as well as alternatives for data.

Detecting data-flow errors in executable BPMN gives way to the following challenges: (1) The BPMN 2.0 specification does not define any formal semantics regarding the execution of flow elements and their data needs and data results. Hence, a transformation of BPMN process models into a formal representation needs to be specified. The transformation has to consider the execution semantics with respect to data elements. (2) Data-flow errors in executable BPMN may have to do with the fact that some data elements are optional, whereas others are mandatory. This leads to additional complexity, compared to the case where everything is mandatory. The definitions of data-flow errors must take all possible combinations into account. (3) A task may read or write data elements out of alternative sets. Detecting errors must take this into account as well. As data elements may be optional at the same time, things become even more complex. (4) There is a lack of publicly available process models conform to BPMN 2.0, which could be used in an evaluation.

This paper proposes an approach to detect data-flow errors in BPMN 2.0 process models. More specifically, we make the following contributions:

Definition of anti-patterns. Starting with classifications of anti-patterns from the literature [10,19,20], we define a set of data-flow anti-patterns tailored to the specifics of data in BPMN 2.0. Our anti-patterns describe anomalies of the data flow. Their definitions consider the execution semantics for data needs and data results of flow elements. In particular, they allow for combinations of alternatives for data elements and distinguish between mandatory and optional data.

Transformation. We specify a transformation of process models into unfolded Petri Nets. The transformation consists of two steps with particular rules. In the first step, the transformation maps a sequence flow with its flow elements to a Petri Net. The second step is an unfolding of data needs and data results into subnets, expressing the execution semantics of flow elements in a formal way. These Petri Nets do so by taking alternatives for data as well as optionality of data into account.

Tool support. We have implemented a tool to detect data-flow errors in BPMN 2.0 process models automatically at design time. It realizes the transformation just mentioned and finds data-flow errors in the Petri Net using a model checker.

Evaluation. We have asked a BPMN expert to develop a set of process models, which we then have used in our evaluation. Our tool detects all data-flow errors systematically generated by him as well as errors occurring in another user experiment.

Using this approach, process designers can now increase the quality of their models by analyzing the data flow of BPMN 2.0 processes at design time. This does away costly process executions with errors. Our approach allows to detect data-flow errors such as *missing* and *redundant optional data*.

Report structure: Section 2 analyzes and explains the data perspective and describes data-flow errors in BPMN 2.0 process models. Section 3 introduces our approach to check data-flow correctness in executable BPMN process models and its implementation. We describe the evaluation of our approach in Section 4. Section 5 discusses Related Work, and Section 6 concludes.

2 Data in BPMN

The BPMN 2.0 standard [13] distinguishes several representations for process models. They differ regarding expressiveness: The *executable subclass* contains the complete execution information. The graphical process diagrams in turn cover only a subset of this. Data elements play different roles in these representations: Data associations to flow elements in graphical process diagrams mean potential data needs and potential data results of flow objects. The specifications for mandatory data, alternatives for data, and optional data in the executable sub-class give precise information on data needs and data results. So-called `InputSet`s and `OutputSet`s, containing `DataInput`s and `DataOutput`s, represent this information, which is not visible in the graphical process diagram.

This design decision of the BPMN 2.0 standard leads on one hand to simple understandable process diagrams for analysts, on the other hand the process diagrams hide information affecting the execution semantics. From the perspective of data-flow correctness, graphical process diagrams are ambiguous.

In the following, we first describe concepts for specifying process diagrams with data graphically and give an example of a BPMN 2.0 process diagram with different data-flow errors. Then we analyze the execution semantics of process flow elements handling data. In the following, we will use the term BPMN instead of BPMN 2.0 if it is unambiguous.

2.1 Data in Graphical Process Diagrams

The BPMN standard [13] provides graphical notations for modeling flow elements in process diagrams. The most important concepts regarding data flow in process diagrams are `DataObject`s and their `DataAssociation`s to the flow elements task and event. A `DataObject`, visualized as document, can be a single

instance or a collection of data elements of the same type, i.e., a `DataObject Collection`. Process designers can define potential data needs and data results by modeling `DataAssociation`s to or from `DataObject`s, visualized as dotted lines. They mean potential reads or writes on the `DataObject`s. All data specifications together determine the graphically visible data flow of a process. BPMN describes `DataObject`s local to the process, so a `DataObject` does not require an explicit deletion. `DataStore`s, `DataInput` and `DataOutput` of processes allow to specify data exchange between databases and processes, thus they do not affect the data flow within the process. Sub-processes may have their own local `DataObjects`, being accessible only within the sub-process. While tasks can have `DataAssociation`s to and from `DataObject`s, events have either `DataAssociation`s to or `DataAssociation`s from `DataObject`s, dependent on their type (catching or throwing).

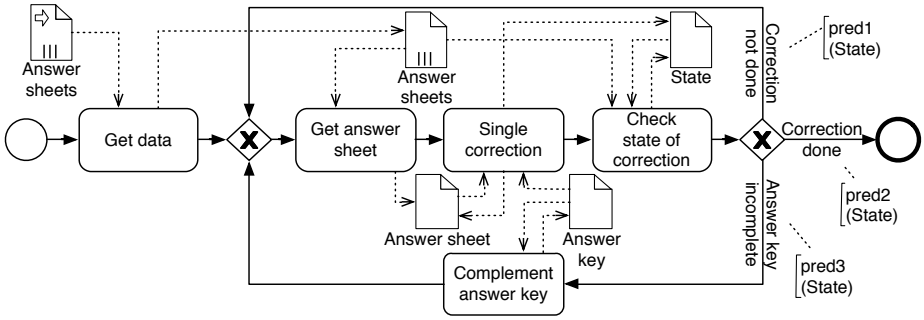


Fig. 1: Exam Correction Process Diagram with Data-Flow Errors

Example 2. Figure 1 displays a process diagram for correcting answer sheets of a written exam. At the beginning, the `DataInput Collection` **Answer sheets** of the process contains a set of answer sheets (i.e., filled out solutions of the exam). Task *Get answer sheet* selects one element of a local copy of this collection and writes it to the `DataObject` **Answer sheet**. A performer of task *Single Correction* marks this **Answer sheet** using correction guidelines given in `DataObject` **Answer key**. Task *Check state of correction* reviews whether all **Answer sheets** have been corrected or not. The task writes its result in `DataObject` **State**. It records the status of the correction process. Dependent on this value, the process continues the correction and turns back to the first gateway (Condition *Correction not done*), or it terminates (Condition *Correction done*). The performer of task *Single correction* may identify a solution in an **Answer sheet** which is not part of the correction guidelines in `DataObject` **Answer key** up to now. In this case, *Single correction* writes the status *Answer key incomplete* to the `DataObject` **State**, and the process runs task *Complement answer key* later on. Otherwise, the **Answer key** already contains the necessary information, and the performer of task *Single correction* does not need to update `DataObject` **State**. Because of this, `DataObject` **State** is optional output of this

task and is written on demand. As the graphical representation does not allow to model optionality of output, this characteristic is not visible in Figure 1.

The process diagram displayed in Figure 1 contains four data-flow errors, which we will describe in Section 2.3.

2.2 Data in Executable Processes

To analyze data flow in BPMN process models, one needs to consider the execution semantics of flow elements handling data. The execution semantics describes the rules for running a process instance, e.g., the pre- and postconditions holding for the *availability* of data for executing flow elements, such as tasks.

To get executable processes, BPMN requires to refine the potential data needs and data results of a task or event, represented as `DataAssociation`s in the process diagram, by a more concrete specification [13]. This is important because there exist several cases of non ambiguous representations in graphical diagrams. For example, several input or output `DataAssociation`s can mean that a task reads or writes all data elements, only one of them, or some combination. Further, process designers can combine specifications for alternatives and optional data.

Example (cont.). In Figure 1, it is unclear whether the data output **State** of task *Single correction* is optional or mandatory.

We now focus on the execution semantics for tasks. The data elements a task requires for its execution are its data needs. When a task is ready for execution, the process engine checks the *availability* of its data needs. The concepts `InputSet` and `OutputSet` describe the data needs and data results. Figure 2 shows an excerpt of our sample process of Figure 1. The boxes display two expanded tasks *Single Correction* and *Check state of correction*, with their `InputSet`s and `OutputSet`s. Each `InputSet` consists of references to `DataInput`, which are associated with one or more `DataObject`. In Figure 2, the ovals with dotted lines represent an `InputSet` and an `OutputSet` of a task (the first ones with small, the latter ones with larger dots), containing `DataInput`s and `DataOutput`s. For the visual representation of `DataInput`s and `DataOutput`s of a *task*, we have harnessed the visual notation of `DataInput` (empty arrow) and `DataOutput` of processes (filled arrow) and have marked it with "T".

An `InputSet` summarizes data needs of a task t, with a mandatory and an optional *subset*. In Figure 2, the parts of the ovals with grey background denote the *optional subset*, and those with white background the *mandatory subset* of `InputSet`s or `OutputSet`s. An `InputSet` $IS(t)$ is *available* if all `DataObject`s referred to in the mandatory subset of the `InputSet` $IS^M(t)$ are available. `DataObject`s referred to in the optional subset of the `InputSet` $IS^O(t)$ do not affect its availability. The converse concept to data needs is data results. They are the output of a task resulting from its execution. Analogously to the `InputSet`s, `OutputSet`s may have mandatory $OS^M(t)$ and optional subsets $OS^O(t)$. In Figure 2, *do_1* is mandatory (white), and *do_2* is optional (grey).

A task may have several `InputSet`s, representing alternative data needs. If so, the process engine takes the first available `InputSet` to execute the task

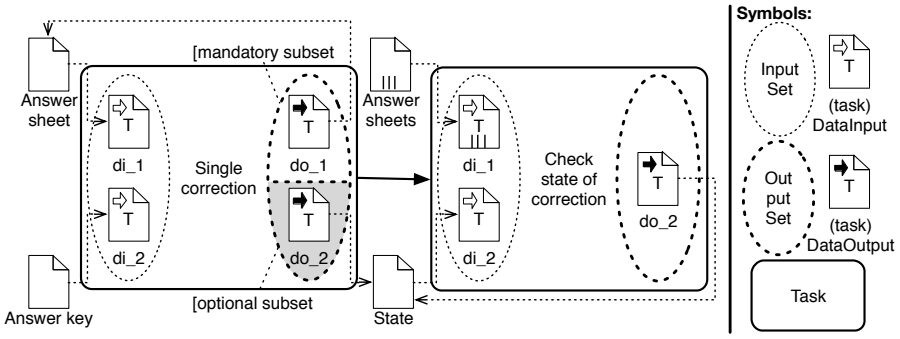


Fig. 2: Data Specifications for two Executable Tasks

according to the order of specification. In other words, at least one `InputSet` has to be available for execution. Analogously to the `InputSet`s, a task may have several `OutputSet`s. When a task terminates, the process uses one `OutputSet` for further execution.

A task is the only flow element which may have both data needs and data results. With respect to events, throwing events may have data needs, and catching events may have data results. The execution semantics of events is slightly simpler than the one of tasks, due to the restriction to only one `InputSet` and one `OutputSet`. Conditional sequence flows can have data-dependent conditions, which are comparable to data needs of tasks and events.

2.3 Data-Flow Errors in BPMN 2.0

A common understanding is that a data flow of a process is correct if there are no anomalies regarding data processed [19]. To capture these anomalies, existing approaches [10,19,20] specify a set of data-flow errors for any data element of a process: *missing data*, *redundant data*, *lost data*, *inconsistent data*, and *wrongly or not destroyed data*. A *missing data* error occurs if a task needs a `DataObject`, but it is not available, because it has not been initialized yet. A *redundant data* error holds if a task writes a `DataObject` which is not read by any task in the subsequent process execution. A *lost data* error happens if a task writes a `DataObject`, and no upcoming task reads it until another task overwrites it. An *inconsistent data* error occurs if one task writes a `DataObject` and another task reads or writes it in parallel. BPMN does not foresee the possibility to delete `DataObject`s explicitly, so *wrongly or not destroyed data* is not relevant in our context. [20] further differentiate between weak and strong variants of redundant and lost data. In other words, it may be some or all execution paths containing the error. Regarding the data aspects of BPMN, these general classifications of errors are relevant as well. However, those publications do not cover the specifics of optionality of data as well as of alternatives for data in BPMN.

Optionality of data. In BPMN a task reads or writes a `DataObject` mandatorily or optionally. This affects the data flow and its correctness. For example,

optional `DataInputs` and optional `DataOutputs` can cause *lost data*, but with partly different effects as in the mandatory case. An optional *lost data* error occurs if a task writes a `DataObject` optionally or mandatorily but it is not read by any task before it is optionally overwritten. This may be problematic, but does not have to be, in contrast to the mandatory case. Furthermore, optional outputs are not sufficient to avoid a missing data error.

Example 3. Our example in Figure 1 contains four data-flow errors: (1 & 2) two *Missing Data* errors, (3) *Weakly Lost Data*, (4) *Strongly Redundant Data*. (1) In the first run of *Single correction*, **Answer key** is uninitialized. (2) Task *Single Correction* initializes **State** only optionally, but *Check state of correction* needs it mandatorily. (3) *Weakly Lost Data* refers to **Answer sheet**. In two execution paths (correction not done, answer key incomplete) there is no task reading **Answer sheet** until *Single Correction* writes it again. (4) There is no task using **Answer sheet** that has been updated by *Single correction*.

Alternatives for data. The data flow of a BPMN process depends on the `InputSets` and `OutputSets` chosen during process execution. This asks for analyzing data flow with respect to these alternatives for data at design time. In each `InputSet` the missing data error depends on the alternatives for `OutputSets` chosen previously. To detect all potentially incorrect data alternatives, we have to consider all possible combinations of `InputSets` and `OutputSets` for each `DataObject` involved. By doing so, we do not have to take the order of `InputSets` and `OutputSets` into account.

Our anti-patterns for BPMN processes reflect both alternatives for data as well as optionality, see Table 1.

3 Detecting Data-Flow Errors

To achieve data-flow correctness in BPMN 2.0 process models, we formalize the execution semantics regarding data elements in BPMN by using a transformation algorithm, see Section 3.1. In Section 3.2 we formalize generic data-flow anti-patterns and say how to generate process-specific anti-patterns for model checking. The final step is proving the possible existence of data-flow errors in the process model with model checking; Figure 3 gives an overview of our approach [14].

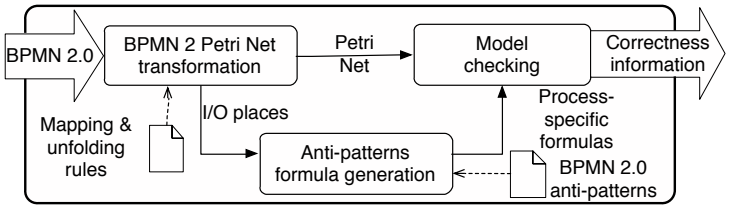


Fig. 3: Overview of Our Approach on Detecting Data-Flow Errors

Algorithm 1: The BPMN2PetriNet Transformation Algorithm

```
Algorithm BPMN2PetriNet()
    for each flow element f_i do
        Map(f_i)                                                    // Step (1)
        if f_i is a data-dependent flow element then
            Unfold(f_i)                                             // Step (2)

Procedure Map(Flow element f)
    Map f to Petri Net                                              // Step (1a)
    Connect f with predecessors & successors                        // Step (1b)

Procedure Unfold(Flow element f)
    if |IS(f)| ≥ 2 ∨ |OS(f)| ≥ 2 then
        Unfold InputSets/OutputSets of f                            // Step (2a)
    for each InputSet/OutputSet IOS_i(f) do
        for each data element d_j ∈ IOS_i(f) do
            Unfold input & output subsets of d_j in IOS_i(f)        // Step (2b)
            Record I/O places of d_j                                // Step (2c)
            if d_j ∈ O_M(IOS_i(f)) ∧ ∃ pred(d_j) then
                Generate predicate subnet of d_j                    // Step (2d)
```

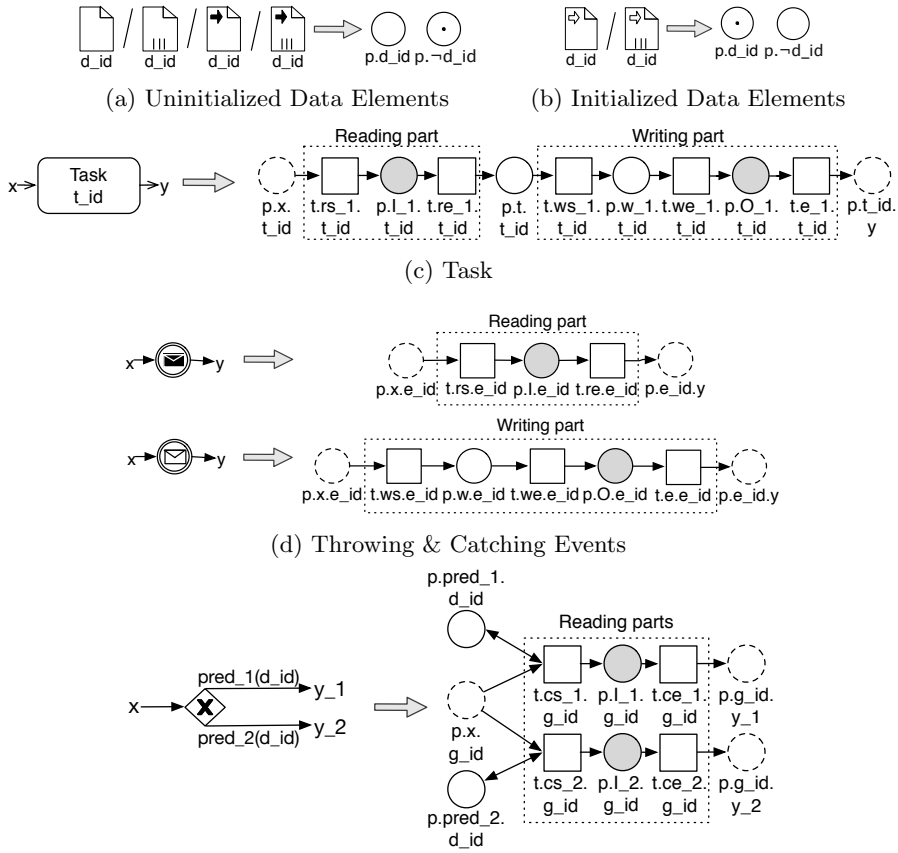
3.1 The BPMN2PetriNets Transformation Algorithm

To analyze process models, many approaches employ Petri Nets to represent the models; see [22] or Appendix A for the definition. For example, [6] uses Petri Nets for representing BPMN 1.0 and [7] for BPEL processes; [8] gives an overview. In a nutshell, we follow the approach in [6] which transforms BPMN 1.0 to Petri Nets. However, [6] does not capture the data characteristics of BPMN 2.0. [2] transforms data objects but in BPMN 1.2. Their way to represent data with Petri Nets including the limited data perspective of BPMN 1.2 is not sufficient for our purposes, and their proprietary interpretation of the execution semantics of data objects and related states is not compatible with BPMN 2.0.

Our BPMN2PetriNet algorithm transforms a BPMN process model into a Petri Net representation in two steps, see Algorithm 1: Step (1) *Mapping* the sequence flow to Petri Nets; Step (2) *Unfolding* the data needs and data results of mapped flow elements.

Mapping. The procedure `Map()` in Algorithm 1 starts with Step (1a) mapping the flow elements, including data elements, to a Petri Net using mapping rules. In Step (1b) it connects the mapped flow element with its successor and predecessor flow elements according to the sequence flow.

The mapping rules distinguish between data elements and other flow elements. To map the sequence flow and data-independent flow elements, e.g., parallel gateways, we use an existing BPMN 1.0 to Petri Nets transformation [6], which requires some preconditions, e.g., for split or join gateways. For all data



(a) Uninitialized Data Elements (b) Initialized Data Elements

(c) Task

(d) Throwing & Catching Events

(e) Conditional Sequence Flows

Fig. 4: Mapping Rules

elements and data-dependent flow elements, i.e., tasks, events, and conditional sequence flows, we need new mapping rules to Petri Net representations. We now describe the mapping rules for data elements and data-dependent flow elements, cf. Figure 4:

Data Elements. The mapping rules for data elements are straight-forward (see Figs. 4a and 4b): The two places $p.d\_id$ and $p.\neg d\_id$, with $m(p.d\_id) + m(p.\neg d\_id) = 1$ number of tokens, represent the initialization status of a `DataObject` or a `DataInput` or `DataOutput` of a process. We distinguish two cases: A `DataObject` or `DataOutput` of a process is uninitialized, see Fig. 4a. A `DataInput` of a process is already initialized, see Fig. 4b.

Tasks. Figure 4c shows the mapping rule for a task. The so-called reading and writing parts serve as interfaces for the subnets generated in the unfolding step and represent one 'empty' `InputSet` and one 'empty' `OutputSet`. But they do not contain their `DataInput` and `DataOutput` elements yet. The unfolding

step will add them to the reading and writing parts. Additionally, the Petri Net representing a mapped task has two *connecting places* $p.x.t\_id$ and $p.t\_id.y$ (dashed lines) for connecting the mapped task with its predecessor flow element x and successor flow element y. Each reading part contains an input place $p.I\_1.t\_id$ and each writing part an output place $p.O\_1.t\_id$ (filled in grey). The input and output places, in short I/O places, in the reading and writing parts are essential for checking the data-flow correctness.

Definition 1 (Input and output places.) *An **input place** $p.I\_i.t\_id$ is a place of the reading part of the i-th* `InputSet` *of a task t_id with the following characteristics: If it has a token, all* `DataInputs` *of the i-th* `InputSet` *have been read successfully. The firing of transition $t.rs\_i.t\_id$ indicates the successful reading. An **output place** is the analogous place of a writing part.*

Events. Due to the restrictions of data needs and data results of events to at most one `InputSet` and one `OutputSet`, the mapping rules for events are less complex than those for tasks (see Figure 4d). A mapped throwing event e_id has only a reading part, a mapped catching event has only a writing part.

Conditional sequence flows. Conditional sequence flows determine which execution path to take after a data-dependent split gateway g_id. We map a data-dependent gateway with outgoing conditional sequence flows as shown in Figure 4e. A mapped data-dependent split gateway g_id consists of one connecting place $p.x.g\_id$ to connect the predecessor of g_id with n outgoing conditional sequence flows. Each of these conditional sequence flows are mapped to a reading part $t.cs\_c$, $p.I\_c.g\_id$ and $t.ce\_c$ and a connecting place $p.g\_id.y\_c$ to its successor flow element y_c. Each mapped conditional sequence flow is connected to the mapped gateway, because place $p.x.g\_id$ is input place of $t.cs\_c.g\_id$. Each of these transitions $t.cs\_c.g\_id$ has an additional input place, a so-called predicate place $p.pred\_c.d\_id$ with $\sum_{c=1}^{n} m(p.pred\_c.d\_id) = 1$. These predicates reflect, according to predicates in [20], the different values a `DataObject` d_id can hold and determine the execution paths. In the following unfolding step the `DataObject` used in this condition is added, according to the rule for a mandatory input subnet (see Figure 6a).

Unfolding. Procedure `Unfold()` of Algorithm 1 adds Petri Net representations of the data needs and data results to an already mapped data-dependent flow element f. It comprises the following steps: Step (2a): The mapped flow element contains one reading part or one writing part, representing one `InputSet` or one `OutputSet`. If the mapped flow element f has several `InputSet`s or `OutputSet`s, we unfold each further `InputSet` $IS_i(f)$ by adding an additional reading part as alternative path to already existing reading parts, i.e., $t.rs\_i.f\_id$, $p.I\_i.f\_id$ and $t.re\_i.f\_id$. For each further `OutputSet` $OS_i(f)$ we add an additional writing part as alternative path to already existing writing parts, i.e., $t.ws\_i.f\_id$, $p.O\_i.f\_id$ and $t.we\_i.f\_id$. Figure 5 shows the resulting Petri Net of a task after unfolding a second `InputSet` and a second `OutputSet`. Further `InputSet`s and `OutputSet`s follow the same scheme.

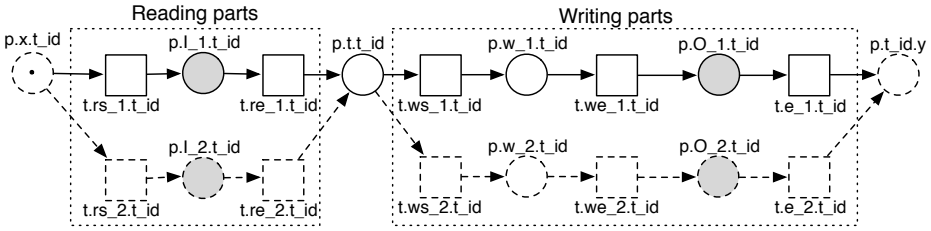
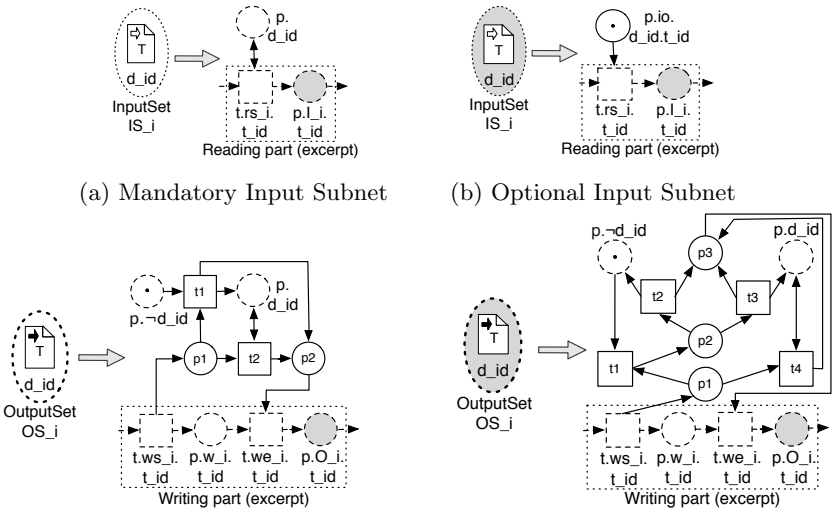


Fig. 5: Task after unfolding a second `InputSet` and a second `OutputSet`s (Step (2a))



(a) Mandatory Input Subnet (b) Optional Input Subnet

(c) Mandatory Output Subnet (d) Optional Output Subnet

Fig. 6: Unfolding Rules

By doing so, we represent alternative `InputSet`s and `OutputSet`s of tasks, leaving aside their ordering. Step (2b): Each `InputSet/OutputSet` has a mandatory and an optional *subset* of `DataInput/DataOutput` elements. Figure 6 represents the *subnets* for these four different cases (input/output, mandatory/optional) for unfolding an element of an `InputSet/OutputSet` $IOS_i(t\_id)$ of a flow element t_id. For each `DataObject` in an `InputSet/OutputSet` we add the appropriate subnet. Next, we connect each unfolded subnet with the reading/writing part of the flow element mapped illustrated by dashed lines in Fig. 6. Step (2c) records all I/O places for the `DataObject` d_j, which we use for the following process-specific anti-pattern formula generation. Regarding Step (2d): If OS_i refers the `DataObject` mandatorily, and if any condition of a conditional sequence flow refers the `DataObject`, we generate a so-called predicate subnet [20]. [20] intro-

duced 'unfolding' of predicate subnets to express that a `DataObject` may hold several values which influence the process execution by data-dependent gateways.

Subnets. Figure 6 summarizes unfolding rules, i.e., the input and output subnets. Our idea behind the subnets is to specify the execution semantics regarding data in BPMN. E.g., the subnet in Fig. 6d covers all three execution scenarios of an optional output d_id of an `OutputSet` $OS(t\_id)$ of a flow element t_id: *(a)* d_id has no value before start of t_id, and t_id initializes d_id, *(b)* d_id has no value before start of t_id and t_id does not write to d_id, and *(c)* t_id writes d_id. The resulting optional output subnet fires as follows: When $p1$ gets a token by transition $t.ws\_j.t\_id$, either $p.d\_id$ or $p.\neg d\_id$ has a token. With it, the subnet may fire as follows: $t1$ fires if places $p.\neg d\_id$ and $p1$ have a token (Cases *(a)* and *(b)*). As a consequence, either $t2$ (Case *(a)*) or $t3$ (Case *(b)*) can fire. Otherwise, the tokens of places $p1$ and d_id enable transition $t4$ to fire (Case *(c)*). With $p3$ the outgoing place of all three 'writing transitions' $t2, t3$, and $t4$, it synchronizes the execution flow of the three paths.

3.2 Formalization of BPMN 2.0 Data-Flow Anti-Patterns

In this section we formalize the data-flow errors as anti-patterns, which we have identified as relevant for a BPMN process model, see Section 2.3. The result is a set of so-called generic anti-patterns that are independent of a concrete process model. Then, we say how to generate process-specific anti-patterns.

Formalizing Generic Anti-Patterns. The rationale of the formalization is as follows. BPMN allows to specify the data needs and data results of flow elements. This results in a data flow from the perspective of an individual `DataObject`. Thus, we examine data-flow errors for each `DataObject` d. To consider alternatives of data modelled as several `Input-` and `OutputSet`s, we analyze the data flow regarding the combinations of these alternatives. Further, supporting mandatory or optional use of data gives way to several data-flow errors we define anti-patterns for. We aim to achieve correctness at design time, however, the availability of data is only known during execution. This is why we have to analyze all possible execution variants that contain execution paths determined by the control flow as well as by the choice of alternative data needs and data results. These variants are contained in the state space of the unfolded Petri Net model which we use for error detection.

We illustrate how to formalize the data-flow errors with anti-patterns, using the example of a *Missing Data* flow error of a `DataObject` d. This error occurs if the process contains a flow element f which needs d mandatorily, and the process has no flow element f' which initializes d mandatorily before the execution of f. f and f' might have several alternative data needs in combination with several alternative data results. Hence, we consider all possible combinations of input and output alternatives where d is used to analyze its data flow.

As a basis for our formalization, for each `DataObject` d we need information on where in the process d is processed (as input or output, optionally or mandatorily). To this end, we now define these sets of BPMN flow elements for a certain `DataObject`. We will make use of these sets to specify the anti-patterns.

For the definitions that follow we assume a process model containing a number of tasks $|tasks|$, a number of events $|events|$, and a number of conditional sequence flows $|conds|$, and label $f = \{tasks\} \cup \{events\}$ the set of all tasks and events of the process model. A task or event f_m has a number of `InputSet`s and `OutputSet`s that we denote with $|IS(f_m)|$ and $|OS(f_m)|$. Note that these sets contain `DataObject`s as data needs and data results. Further, we assume that the process model contains conditional sequence flows s and respective conditions $cond(s)$ with a number of $|cond|$. A conditional sequence flow has no alternative data uses, i.e., there is only one `InputSet` with mandatory data elements for each $cond(s)$. Let $IS_i^O(f_m)$ be the optional subset of the i-th `InputSet` IS_i of the m-th task or event, and $OS_i^O(f_m)$ the optional subset of the i-th `OutputSet` OS_i of the m-th task or event; an event has at most one `InputSet` or one `OutputSet`.

In the following definition, we summarize all `InputSet`s $Set\_I_O(d)$ resp. `OutputSet`s $Set\_O_O(d)$ d is an optional element of. Using these sets, $I_O(d)$ specifies if d has been read successfully in one of the `InputSet`s of $Set\_I_O(d)$; correspondingly, $O_O(d)$ specifies if d has been written successfully in one of the `OutputSet`s of $Set\_O_O(d)$.

Definition 2 (Optional `InputSets`/`OutputSets` containing `DataObject` d)

$$Set\_I_O(d) = \bigcup_{m \in \{1..|f|\}} \{IS_i(f_m) : d \in IS_i^O(f_m) \mid i \in \{1..|IS(f_m)|\}\}$$

$$Set\_O_O(d) = \bigcup_{m \in \{1..|f|\}} \{OS_i(f_m) : d \in OS_i^O(f_m) \mid i \in \{1..|OS(f_m)|\}\}$$

$$read(x) = \textit{ all } \texttt{DataInputs} \textit{ in } \texttt{InputSet } x \textit{ have been read successfully}$$

$$written(x) = \textit{ all } \texttt{DataOutputs} \textit{ in } \texttt{OutputSet } x \textit{ have been written successfully}$$

$$I_O(d) = \bigvee_{x \in Set\_I_O(d)} \big(read(x)\big) \qquad O_O(d) = \bigvee_{x \in Set\_O_O(d)} \big(written(x)\big)$$

For the mandatory use of d, we specify $I_M(d)$ and $O_M(d)$ analogously to Definition 2. In addition to tasks and events, a conditional sequence flow s can use a `DataObject` within a condition $cond(s)$ mandatorily as well. The definitions of $I_M(d)$ and $O_M(d)$ are according to the optional case.

In the following definition, we summarize all `InputSet`s $Set\_I_M(d)$ resp. `OutputSet`s $Set\_O_M(d)$ d is a mandatory element of. Using these sets, $I_M(d)$ specifies if d has been read successfully in one of the `InputSet`s of $Set\_I_M(d)$; correspondingly, $O_M(d)$ specifies if d has been written successfully in one of the `OutputSet`s of $Set\_O_M(d)$.

Definition 3 (Mandatory `InputSets`/`OutputSets` containing `DataObject` d)

$$Set\_I_M(d) = \{s_l : d \in cond(s_l) \mid l \in \{1..|cond|\}\} \cup \Big(\bigcup_{m\in\{1..|f|\}} \{IS_i(f_m) : d \in IS_i^M(f_m) \mid i \in \{1..|IS(f_m)|\}\}\Big)$$

$$Set\_O_M(d) = \bigcup_{m\in\{1..|f|\}} \{OS_i(f_m) : d \in OS_i^M(f_m) \mid i \in \{1..|OS(f_m)|\}\}$$

$$read(x) = \textit{ all } \texttt{DataInputs} \textit{ in } \texttt{InputSet } x \textit{ have been read successfully}$$

$$written(x) = \textit{ all } \texttt{DataOutputs} \textit{ in } \texttt{OutputSet } x \textit{ have been written successfully}$$

$$I_M(d) = \bigvee_{x\in Set\_I_M(d)} \big(read(x)\big) \qquad O_M(d) = \bigvee_{x\in Set\_O_M(d)} \big(written(x)\big)$$

We now formalize the anti-patterns for BPMN using a temporal logic formalism, namely CTL (**C**omputation **T**ree **L**ogic). See [5], Appendix A for a description of CTL. [5] introduces an effective algorithm to verify properties specified in CTL on Petri Net models. To achieve data-flow correctness, our approach aims to identify errors which occur during reading or writing of a `DataObject` in the context of a certain choice of alternatives. Next, the specification of data needs and results in BPMN allows for optional or mandatory use of data. This gives way to a distinction between optional and mandatory errors, as described in Section 2.3. Further, we differentiate between *weak* and *strong* variants of redundant and lost data, see Section 2.3. The result is a set of *generic anti-patterns* for a `DataObject` d. Table 1 lists our generic anti-patterns tailored to the execution semantics of BPMN.

Anti-Patterns - DAP	**Formalization**
1 Missing Data	$\mathrm{E}\big(\neg O_M(d)\ \mathrm{U}\ I_M(d)\big)$
2 Missing Optional Data	$\mathrm{E}\big(\neg(O_O(d) \vee O_M(d))\ \mathrm{U}\ I_O(d))\big)$
3 Strongly Redundant Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{AX}(A[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ \mathrm{term}])\big)$
4 Weakly Redundant Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{EX}(E[\neg(I_M(d) \vee I_O(d)\ \mathrm{U}\ \mathrm{term}]\big)$
5 Redundant Optional Data	$\mathrm{EF}\big(O_O(d) \wedge \mathrm{EX}(E[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ \mathrm{term}])\big)$
6 Strongly Lost Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{AX}(A[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)])\big)$
7 Weakly Lost Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{EX}\big(E[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)]\big)\big)$
8 Lost Optional Data	$\mathrm{EF}\big(O_O(d) \wedge \mathrm{EX}\big(E[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)]\big)\big)$
9 Optional Lost Data	$\mathrm{EF}\big(\big(O_M(d) \wedge \mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_O(d)]\big)$ $\vee\big(O_O(d) \wedge \mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_O(d)]\big)\big)$
10 Inconsistent Data	$\bigvee_{f\in\{E\cup T\}:d\in OS(f)} \mathrm{EF}\big[exec(f) \wedge \bigvee_{f'\neq f:d\in OS(f)} exec(f')\big]$

Legend (for CTL formulas):
A path operator (A or E) occurs together with a state operator (X, F, U).
A/E: the formula needs to hold in *all/at least one of the* succeeding execution paths,
X/F: the formula holds in the *next/at least in one succeeding* state,
$[\phi_1\ \mathrm{U}\ \phi_2]$: ϕ_1 holds until ϕ_2 is reached.

Table 1: BPMN 2.0 Generic Anti-Patterns for a `DataObject` d

In the following, we describe our formalized anti-patterns.

DAP 1: Missing Data occurs if a `DataObject` d is mandatory input $I_M(d)$, but not a mandatory output $O_M(d)$ before, i.e., d is not initialized.

DAP 2: Missing Optional Data means that a `DataObject` d is optional input $I_O(d)$, and d is not initialized before by a mandatory output $O_M(d)$ or by an optional output $O_O(d)$.

DAP 3: Strongly Redundant Data is given if a `DataObject` d is a mandatory output $O_M(d)$, but no flow element uses this `DataObject` as mandatory input $I_M(d)$ or as an optional input $I_O(d)$ in all following execution paths.

DAP 4: Weakly Redundant Data happens if there exist at least one execution path of the process, in which a `DataObject` d is neither a mandatory input $I_M(d)$ nor an optional input $I_O(d)$, but `DataObject` d is a mandatory output $O_M(d)$ before.

DAP 5: Redundant Optional Data holds for a `DataObject` d, if an optional output $O_O(d)$ is not used in the future, namely `DataObject` d is no mandatory input $I_M(d)$ or optional input $I_O(d)$ in one of the succeeding execution paths.

DAP 6: Strongly Lost Data occurs if a `DataObject` d is several times mandatory output $O_M(d)$, but before the later mandatory outputs happen, there is no mandatory input $I_M(d)$ in between. This situation holds for all execution paths and means that earlier data results (e.g., initializations, updates) for d get lost.

DAP 7: Weakly Lost Data means that a `DataObject` d is mandatory output $O_M(d)$, and there exists at least one execution path where it is mandatory output $O_M(d)$, but not mandatory input $I_M(d)$ before.

DAP 8: Optional `DataOutput` does not lead to an error in every case. *Lost Optional Data* are `DataObject`s d which are optional ($O_O(d)$) or mandatory output ($O_M(d)$). However, no upcoming flow elements exist that read d ($I_M(d) \vee I_O(d)$) until a flow element again writes d optionally, i.e., U $O_O(d)$.

DAP 9: Optionally Lost Data includes all `DataObject`s d which are mandatory ($O_M(d)$) or optional output ($O_O(d)$). Additionally, there is a flow element that optionally writes d subsequently without a flow element reading d in between (optionally or mandatorily).

DAP 10: Inconsistent Data Two elements are in conflict if both want to read d, and one element also writes d, i.e. d belongs to its data results. f must write d optionally or mandatorily, i.e., $f \in O_O(d) \cup O_M(d)$. d is inconsistent if two elements in conflict regarding d can be executed in parallel.

Figure 7 shows process models which illustrate the ten different anti-patterns of Table 1. The models are small examples for the undesired behavior of data-flow for each anti-pattern. Some of these models include more than one data-flow error, e.g., the process model in Figure 7f illustrates strongly lost data but in addition it includes a strongly redundant data-flow error, because the *Data Object* is not read by any task before the process terminates. Furthermore, each strong data-flow error is also a weak one.

Generating and Checking Process-Specific Anti-Patterns. Using our generic BPMN data-flow anti-patterns, the Step *Anti-patterns formula generation* of

our algorithm in Figure 3 delivers process-specific formulas of the generic anti-patterns for each `DataObject`. To this end, we use the results of the transformation from BPMN to Petri Nets with its *Input and Output Places*, see Definition 1. The generation step of process-specific anti-patterns instantiates the generic anti-patterns by means of the optional and mandatory usages of a `DataObject` d according to Definition 2. To check if d is an optional/mandatory data input, we have to prove that an $IS \in Set\_I_O(d) / \in Set\_I_M(d)$ is available. To do so, we have to check if the input place of IS contains a token, see Definition 1. This means that $read(IS)$ (an `InputSet` of a flow element t_id) is replaced with $\mathrm{m}(p.I\_i.t\_id) = 1$. Accordingly for the `OutputSet`s, $written(t\_id)$ is replaced using the *output place*, i.e., with $m(p.O\_i.t\_id) = 1$.

For the final step, we employ a representation of the possible states of the process model to find the states with errors. A model checker, namely LoLA [17], analyzes the process-specific formulas of the anti-patterns for each `DataObject` on the unfolded Petri Net model in question. If a formula is satisfied, the data-flow error is detected.

3.3 Tool Support

We have implemented a tool for automatically analyzing data-flow errors in process models specified with BPMN, see Figure 3. In more detail, it maps all flow elements of the process model and unfolds the `InputSet`s and `OutputSet`s applying our transformation rules, see Figure 4 and Figure 6. The current implementation supports one `InputSet` and one `OutputSet` of a flow element using the anti-patterns defined. Then, for each `DataObject` d of the process model our algorithm generates process-specific anti-pattern formulas considering the actual I/O places of d. The transformed process model, i.e., the resulting unfolded Petri Net, and the process-specific anti-pattern formulas are input to a model checking tool, namely LoLA [17]. LoLA is used to create the state space of the Petri Net generated and to verify the process-specific anti-patterns formulas for each `DataObject` of the process model on this state space.

Our focus is on the data-flow perspective of BPMN; therefore, our tool analyzes all data-flow errors. Checking the control flow, e.g., soundness, is left to existing tools, e.g., ([22]). Common approaches to check control flow often require a Petri Net representation of the process, so that a coupling with our approach is straightforward.

Our evaluation makes use of this tool, see Section 4. The model checking step has run effectively, i.e., in few seconds, for all process models analyzed. A problem could arise if the number of states in the state space becomes too large. This can result from having multiple parallel writings of data objects. For capturing this problem see [12] for a possible reduction technique of the state space.

4 Evaluation

The evaluation of our approach consists of several steps from designing a collection of process models with intensive use of data needs and data results, explicitly adding data-flow errors into some of the process models to a user experiment for modeling data in process models given.

Step 1 is auxiliary, to prepare the evaluation of Steps 2 and 3. To evaluate process models of realistic complexity, we have faced the problem that such a set of processes we could use as benchmark was not yet available publicly. To deal with this problem, we asked an expert to design process models for 11 scenarios [24]. These scenarios use data intensively. Some examples are adapted from literature, others we have developed ourselves. The scenarios comprise, say, order handling (S1) and job interview (S7) and are available online, see http://dbis.ipd.kit.edu/2134.php. These process models do not have any data-flow errors according to our definition. We also checked these process models (namely the Processes Sx.1 in Table 1) with our data-flow analysis tool, which confirmed their error-freeness. Sx.n stands for the n-th variant of a process model of Scenario x. With our tool we have also successfully analyzed the motivating example process, see Section 2.1.

In Step 2, our expert has added data-flow errors to some of the error-free process models. These process models cover all types of data-flow errors (see S3.2, S5.2, S8.2, S9.2 and S10.2 in Table 1). Our data-flow analysis tool has correctly detected all of them.

In Step 3, we have run a user experiment to understand the difficulties of modeling an error-free data flow, and also to obtain process models with data-flow errors for further evaluation of our tool. We have organized this experiment as an exercise of a lecture with seven students with knowledge in BPMN modeling, including the data aspect. These experienced individuals have started with two process models given, namely S1.1 and S2.1 (from Step 1 of our evaluation). We had removed the data elements from the models before. The task has been to enhance the process models with data needs and results. Modifying them has also been allowed if necessary. We textually described the use of data, so that the participants were able to model the data perspective of the processes. We have obtained several process models for the two scenarios. They differ in the number of `DataObject`s and data-dependent flow elements, see Scenarios S1 & S2 in Table 2. With our tool, we then have checked the models. Our tool has detected all data-flow errors contained in the models, 40 errors altogether.

Table 2 gives an overview of the results of our evaluation, i.e., detecting data-flow errors and confirming the correctness of process models without data-flow errors. Sx.n denotes the n-th process of Scenario x. We list the size of the BPMN process models analyzed, the size of the Petri Nets generated and the number of data-flow errors identifed. The number of the BPMN elements determines the size of the corresponding Petri Net, defined by the number of transitions and places. The input and output subnets in particular, added in the unfolding step of our transformation, increase the size of the Petri Net.

Process	# Tasks	# Events	# XOR Splits	# conditions	# DataObjects	# DataAssoc.	# Places	# Transitions	# Missing Data	# Redundant Data	# Lost Data	# Inconsistent Data
S1.1	6	4	1	2	8	12	62	43	4	1	0	0
S1.2	6	4	1	2	5	10	56	43	1	0	0	0
S1.3	6	4	1	2	3	6	56	41	1	2	0	0
S1.4	6	4	1	2	4	9	54	41	1	1	0	0
S1.5	6	4	1	2	4	11	62	43	2	0	0	0
S1.6	6	4	1	2	4	5	49	40	4	1	0	0
S1.7	6	4	1	2	4	7	58	41	3	4	0	0
S1.8	6	4	1	2	8	16	72	50	0	0	0	0
S2.1	8	2	1	2	10	21	79	55	1	0	0	0
S2.2	8	2	1	2	6	13	71	51	1	0	0	0
S2.3	8	2	1	2	5	12	63	47	1	2	0	0
S2.4	8	2	1	2	4	13	63	51	3	3	0	0
S2.5	8	2	1	2	9	21	84	58	0	0	0	0
S2.6	7	2	1	2	9	14	75	49	2	3	1	0
S3.1	8	2	3	6	5	17	85	74	0	0	0	0
S3.2	8	2	2	4	5	15	80	70	0	2	1	0
S4.1	14	2	0	0	12	24	107	82	0	0	0	0
S5.1	7	2	1	2	5	15	64	52	0	0	0	0
S5.2	7	2	1	2	6	16	64	52	0	0	2	0
S6.1	8	2	2	4	7	17	90	70	0	0	0	0
S7.1	6	2	1	2	5	10	58	33	0	0	0	0
S8.1	13	3	2	5	8	22	110	94	0	0	0	0
S8.2	14	3	2	5	9	16	121	104	1	5	5	2
S9.1	8	2	1	2	3	21	72	58	0	0	0	0
S9.2	8	2	1	2	5	24	79	62	0	2	2	0
S10.1	8	2	2	4	5	12	75	60	0	0	0	0
S10.2	8	2	2	5	6	13	91	81	0	1	0	0
S11.1	19	3	2	5	6	13	91	81	0	0	0	0

Table 2: Evaluation of the Correctness Tool

For one, the experiment of Step 3 (user experiment) shows that our tool has detected all data-flow errors in the process models created. Further, *Missing Data* is a frequent error in process modeling. The numbers of *Redundant Data* errors and of *Lost Data* errors reflect that we count both strong as well as weak variants of data-flow errors. *Inconsistent Data* errors occur when different tasks read or write a `DataObject` in parallel. This only happens with parallel execution paths. Only one of our scenarios (S8.2) has this characteristic. All in all, the evaluation shows that in process modeling all types of data-flow errors are relevant and can occur, and our tool has detected all of them.

5 Related Work

Behavioural analysis of process models without the data aspect has been studied extensively, see [9] for an overview. In what follows, we focus on the correctness of the data flow. [16] is one of the first approaches illustrating the importance of data-flow correctness in process modeling. The authors have thoroughly analyzed problems which can occur with a data flow but do not provide a solution for error detection. [3] defines data-flow errors as patterns, but focuses on visually specifying compliance rules in order to explain the violations. There, key requirements are, say, availability of data input and data output of an activity, and consistent flow of data between activities. We in turn use the patterns to express the execution semantics of process models with data elements and thus to analyze the correctness of the data flow. [21] proposes using patterns for the analysis of general compliance violations. In particular, order and occurrence patterns support the users when specifying constraints on a process model with data. However, they do not support BPMN but use BPEL with its specific data semantics for process modeling.

[20] introduces a method based on CTL* that combines the detection of control-flow and data-flow errors. They use anti-patterns for missing data, inconsistent data, redundant data, and lost data for Workflow-Net process models. In contrast to our approach, they do not cover the BPMN 2.0 semantics of data during process execution, including the specific ways to use data, i.e., alternatives for data and mandatory as well as optional data. [19] regards data-flow analysis in processes based on UML activity diagrams. The authors provide separate procedural correctness proofs for each of the three basic types of data-flow errors, namely missing data, conflicting data, and redundant data. [10] extends the results of [19] for UML activity diagrams, by discussing additional data-flow errors such as inconsistent data. For error detection they also use separate checking procedures for each error type. To do so, they explicitly generate and store all possible paths of the process model (particularly due to XOR-Nodes). Next, they do not use a state-based approach to represent the dynamic behaviour of processes (e.g., with a state space of a Petri Net). Further approaches exist using UML data-flow analysis on UML activity models, e.g., [18], and [23], with another focus than ours. [15] deals with data-flow correctness of BPMN 2.0. They use the work of [19] adding optional reading and writing access. To this

end, they add behavioral profiles consisting of information on conflicts between a pair of nodes of a process model. In contrast to our method, their modeling of errors with behavioral profiles handles data separately from the process model and they do not take alternatives into account.

Considering more intricate dependencies of data objects in data-aware process models is subject of further approaches. For instance, some deal with data dependencies like inclusion, referential dependencies [11], semantically defined constraints [4] or with information leaks [1]. These approaches focus on dependencies which are not part of the BPMN data perspective and would require to enhance its specification concepts. In other words, the problem is different from the one studied here.

6 Conclusions

In this paper, we have proposed a new method for detecting data-flows errors in BPMN 2.0 process models at design time. This approach takes alternatives for data as well as optional data into account. An automatic detection scheme requires a formal representation of the execution semantics of BPMN 2.0 flow elements with data associations. To achieve this, we have developed transformation rules and a set of anti-patterns representing data-flow errors in BPMN 2.0 process models. On this basis, we transform data-dependent flow elements of a process model into unfolded Petri Nets to detect data-flow errors by using an existing model checker. Experiments with users have shown that our tool identifies the data-flow errors present.

References

1. R. Accorsi and A. Lehmann. Automatic Information Flow Analysis of Business Process Models. In *BPM*, 2012.
2. A. Awad, G. Decker, and N. Lohmann. Diagnosing and Repairing Data Anomalies in Process Models. In *BPM Workshops*, 2010.
3. A. Awad, M. Weidlich, and M. Weske. Visually Specifying Compliance Rules and Explaining Their Violations for Business Processes. *Visual Languages and Computing*, 22(1), 2011.
4. D. Borrego et al. Diagnosing Correctness of Semantic Workflow Models. *Data & Knowledge Engineering*, 87, sep 2013.
5. E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In *TOPLAS*, 1986.
6. R. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology*, 50(12), 2008.
7. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *BPM*, 2005.
8. N. Lohmann, E. Verbeek, and R. Dijkman. Petri Net Transformations for Business Processes – A Survey. In *Transactions on Petri Nets and Other Models of Concurrency II*. 2009.
9. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer, 1992.

10. H. S. Meda, A. K. Sen, and A. Bagchi. On Detecting Data Flow Errors in Workflows. *Data and Information Quality*, 2(1), 2010.
11. A. Meyer et al. Modeling and Enacting Complex Data Dependencies in Business Processes. In *BPM*, 2013.
12. R. Mrasek. Überprüfung von Fahrzeug-Testworkflows mittels einer Regelbasis in temporaler Logik (in German). Master's Thesis, KIT, April 2013.
13. Object Management Group. Business Process Model and Notation, V2.0. OMG Specification, 2011.
14. Putze, S. Modeling and Analysis of Data in Business Process Models. Master's Thesis, KIT, February 2013.
15. A. Rogge-Solti et al. Business Process Configuration Wizard and Consistency Checker for BPMN 2.0. In *BPMDS*, 2011.
16. S. Sadiq et al. Data Flow and Validation in Workflow Modelling. In *Australian Database Conference*, 2004.
17. K. Schmidt. LoLA a Low Level Analyser. In *Appl. and Theory of Petri Nets*, 2000.
18. H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes Theor. Comput. Sci.*, 127(4), 2005.
19. S. X. Sun et al. Formulating the Data-Flow Perspective for Business Process Management. *Information Systems Research*, 17(4), 2006.
20. N. Trčka, W. v. d. Aalst, and N. Sidorova. Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows. In *CAISE*, 2009.
21. W.-J. van den Heuvel, A. Elgammal, and O. Turetken. Using Patterns for the Analysis and Resolution of Compliance Violations. *Coop.Inf.Systems*, 21(1), 2012.
22. W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Circuits, Systems and Computers*, 8(1), 1998.
23. T. Waheed, M. Iqbal, and Z. Malik. Data Flow Analysis of UML Action Semantics for Executable Models. In *Europ. Conf. Model Driven Architecture*, 2008.
24. Zink, M. Datenspezifikation in BPMN 2.0 (in German). Bachelor's Thesis, KIT, February 2014.

A Basic Concepts for Process Analysis

In this appendix, we review fundamental concepts for data-flow analysis in processes, namely Petri Nets with its state space formalism and the temporal logic CTL.

Petri Nets and State Space Formalism. Petri Nets are a representative of formal graph-based process languages. The definition of a Petri Net used here is the one from [22]. A Petri Net is a directed bipartite graph with two types of nodes called places and transitions.

Definition 4 (Petri Net) *A Petri Net is a triple* (P, T, F) *with* P *a set of places,* T *a set of transitions* $(P \cap T = \emptyset)$ *and* $F \subseteq (P \times T) \cup (T \times P)$ *a set of arcs (flow relation).*

$p \in P$ is an input place of $t \in T$ if $(p, t) \in F$ and an output place if $(t, p) \in F$. $\bullet t$ denotes the set of input places of t and $t\bullet$ the set of output places. A mapping $M : P \to \mathbb{N}_0$ maps every $p \in P$ to a positive number of tokens, i.e., at any time a place contains zero or more *tokens*. The distribution of tokens over places (M) represents a state of the Petri Net, often referred to as its *marking*. A transition $t \in T$ is activated in a state M if $\forall p \in \bullet t : M(p) \geq 1$. A transition $t \in T$ in M can fire, leading to a new state M' which reduces the value of $M(p)$ by 1 if $p \in \bullet t$, adds 1 to $M(p)$ if $p \in t\bullet$ and does not change otherwise. The set of reachable states from a start state M_0 of a Petri Net build the *state space*. To check properties of a BPMN process, we need this state space, for which we use the Kripke structure [20] of the Petri Net corresponding to the original BPMN model.

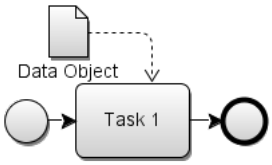
CTL. **C**omputation **T**ree **L**ogic *CTL* is a temporal logic formalism often used to specify properties for model checking. In our case, those properties are data-flow anti-patterns. E.g., [5] describes CTL and an effective algorithm to verify properties specified in CTL.

The formal syntax of CTL is as follows:

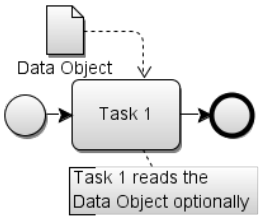
Definition 5 (Computation Tree Logic) *Every atomic proposition ap is a CTL formula. If* ϕ_1 *and* ϕ_2 *are CTL formulas then* $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\mathsf{AX}\phi_1$, $\mathsf{EX}\phi_1$, $\mathsf{AG}\phi_1$, $\mathsf{EG}\phi_1$, $\mathsf{AF}\phi_1$, $\mathsf{EF}\phi_1$, $\mathsf{A}[\phi_1 \ \mathsf{U} \ \phi_2]$, $\mathsf{E}[\phi_1 \ \mathsf{U} \ \phi_2]$ *are CTL formulas.*

In our context, *ap* is a state of a Petri Net which represents the status of a data element in the BPMN process.

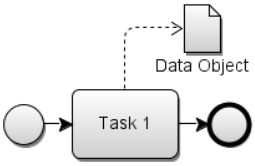
The logical operators always occur in pairs: A path operator (A or E) together with a state operator (X,G, F or U). A means that the formula needs to hold in *all* succeeding execution paths. E means that at least one execution path *exists* where the formula holds. X means that the formula holds in the next state, G means the formula holds in all succeeding states, F means that the formula holds at least in one succeeding state, $[\phi_1$ U $\phi_2]$ means that ϕ_1 holds until ϕ_2 is reached.



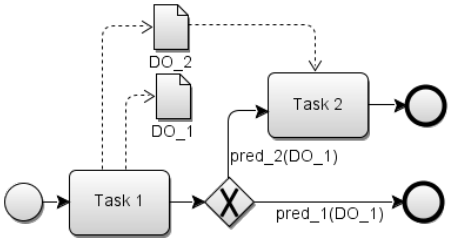
(a) DAP 1 - Missing Data



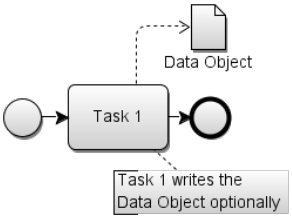
(b) DAP 2 - Missing Optional Data



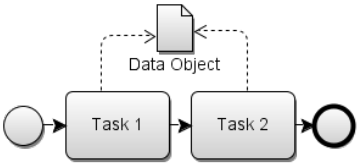
(c) DAP 3 - Strongly Redundant Data



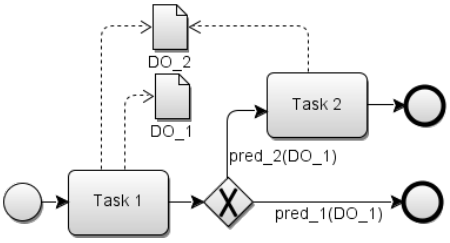
(d) DAP 4 - Weakly Redundant Data



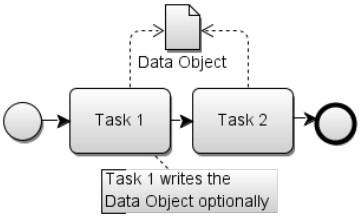
(e) DAP 5 - Redundant Optional Data



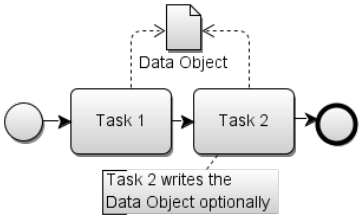
(f) DAP 6 - Strongly Lost Data



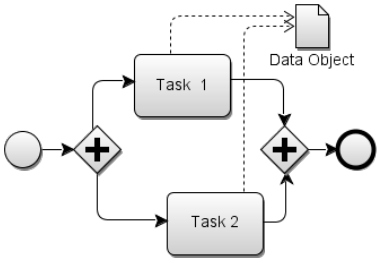
(g) DAP 7 - Weakly Lost Data



(h) DAP 8 - Lost Optional Data



(i) DAP 9 - Optionally Lost Data



(j) DAP 10 - Inconsistent Data

Fig. 7: Examples of Process Models Illustrating all Anti-Patterns of Table 1